Sorting

Recursion and Divide and Conquer

Recursion

- A recursive function is a function that calls itself

```
void recfun()
{
    recfun();
}
```

- Can also say “a function that is defined in terms of itself is recursive”

Two fundamental rules of recursion

- 1) **base case**. Must always have some base case, which can be solved without recursion
- 2) **making progress**. For the cases to be solved recursively, the recursive call must always make progress *towards* a base case

```
void recfun()
{
    recfun();
}
```

`recfun()` has no base case and makes no progress. `factorial()` has a base case and makes progress.

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Tail recursion

- Tail recursion: a recursive call on the last line of the function
- Tail-recursive functions can always be re-written using no recursion
  - Enclose the body of the function in a ‘while’ loop
  - Replace the recursive call with one assignment per function argument
- Non-recursive implementations are generally faster (doesn’t need call stack) but they are also ‘less clear’ for programmers to read/understand
- In tail recursion, nothing needs to be saved (can just return to top of loop)

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

```
int n = 4;
int val = n;

while (n > 1)
{
    val = val * (n - 1);
    n -= 1;
}
```

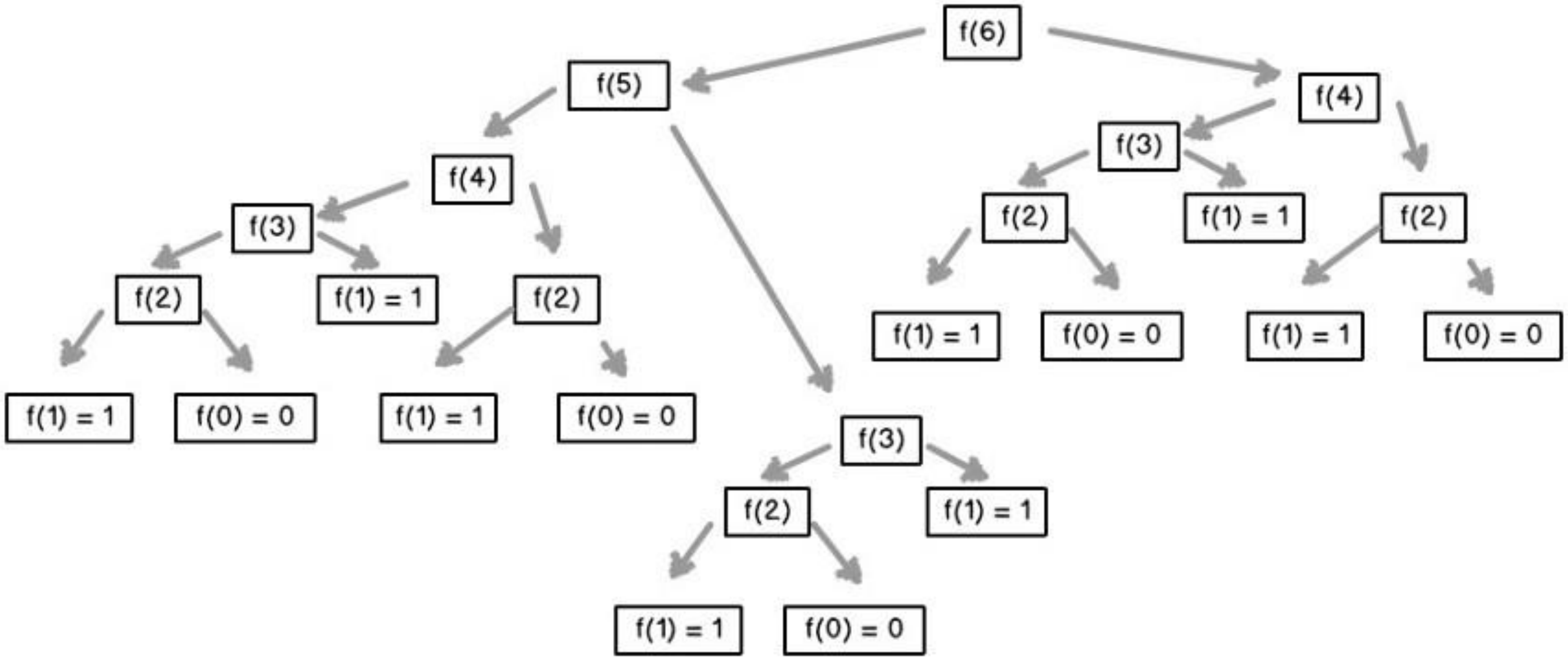
Example of when *not* to use recursion

- Computing Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc..
- Fibonacci sequence def:
- $F_0 = 0,$
- $F_1 = 1,$
- $F_n = F_{n-1} + F_{n-2}$

```
int recfib(int n)
{
    if (n == 1 || n == 0)
        return n;
    else
        return recfib(n-1) + recfib(n - 2);
}
```

Recursive fibonacci call tree

- Time complexity of recursive fibonacci implementation is $O(2^n)$
- https://stackoverflow.com/questions/360748/computational-complexity-of-fibonacci-sequence



Iterative Fibonacci

- Using a vector
- Not using a vector
- Both are $O(n)$

```
int fib[] = {1,1};
for (int i = 3; i < fibnum; ++i)
{
    int temp = fib[1];
    fib[1] = fib[0] + fib[1];
    fib[0] = temp;
}
```

```
int fibnum = 7;
std::vector<int> sums{ 0,1 };
for (int i = 2; i <= fibnum; ++i)
{
    sums.push_back(sums[i - 1] + sums[i - 2]);
}
```

Binary search

- Binary search: algorithm to find a given element in a sorted array
- Example: a = [1,2,3,4,5,6,7,8]
- Recursive implementation:

```
int binarySearch(std::vector<int>& a, int elem, int startInd, int endInd)
{
    if (startInd == endInd && a[startInd] != elem)
        return -1;
    else if (a[startInd] == elem)
        return startInd;
    else {
        int middleInd = (startInd + endInd) / 2;
        if (elem <= a[middleInd])
            return binarySearch(a, elem, startInd, middleInd);
        else
            return binarySearch(a, elem, middleInd+1, endInd);
    }
}
```

Binary search (non-recursive)

```
std::vector<int> a{1,2,3,4,5,6,7,8};

int foundIndex = 0;
int startIndex = 0;
int endIndex = a.size() - 1;
int elemToFind = 8;

while (startIndex != endIndex)
{
    int middleIndex = (startIndex + endIndex) / 2;
    if (elemToFind <= a[middleIndex])
        endIndex = middleIndex;
    else
        startIndex = middleIndex + 1;
}
if (a[startIndex] == elemToFind)
    foundIndex = startIndex;
else foundIndex = -1;
```

Divide and Conquer

- Divide and conquer approach (3 steps)
- 1 – **divide** the problem into a number of subproblems
- 2 – **conquer** the subproblems by solving them recursively
  - If the subproblems are small enough, solve them in a straightforward manner
- 3 – **combine** solutions to subproblems into solution for original problem

- Binary search is an example of **decrease and conquer**, *not* divide and conquer

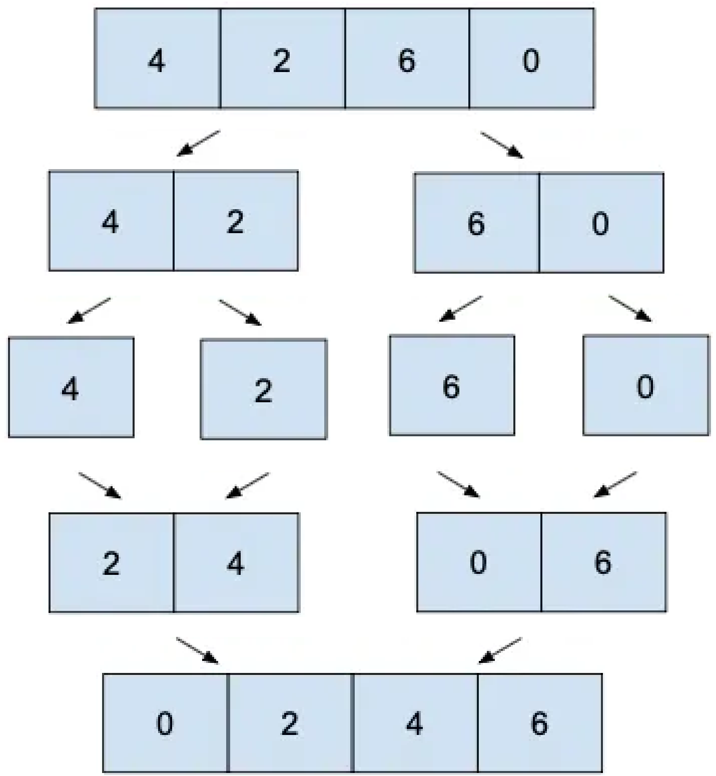
Merge sort

- **Merge sort** uses the divide and conquer paradigm to sort a sequence of items
- 1) **divide**: divide the n-element sequence into two subsequences of $n/2$ elements each
- 2) **conquer**: sort the two subsequences recursively using merge sort
- 3) **merge** the two sorted subsequences to produce sorted result

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Merge sort

```
A = [4,2,6,0]
Merge-sort(A,0,3)
Merge-sort(A,0,1)
Merge-sort(A,0,0)
Merge-sort(A,1,1)
Merge(A,0,0,1)
Merge-sort(A,2,3)
Merge-sort(A,2,2)
Merge-sort(A,3,3)
Merge(A,2,2,3)
Merge(A,0,1,3)
```



MERGE-SORT(A, p, r)

if $p < r$

  then $q \leftarrow \lfloor (p+r)/2 \rfloor$

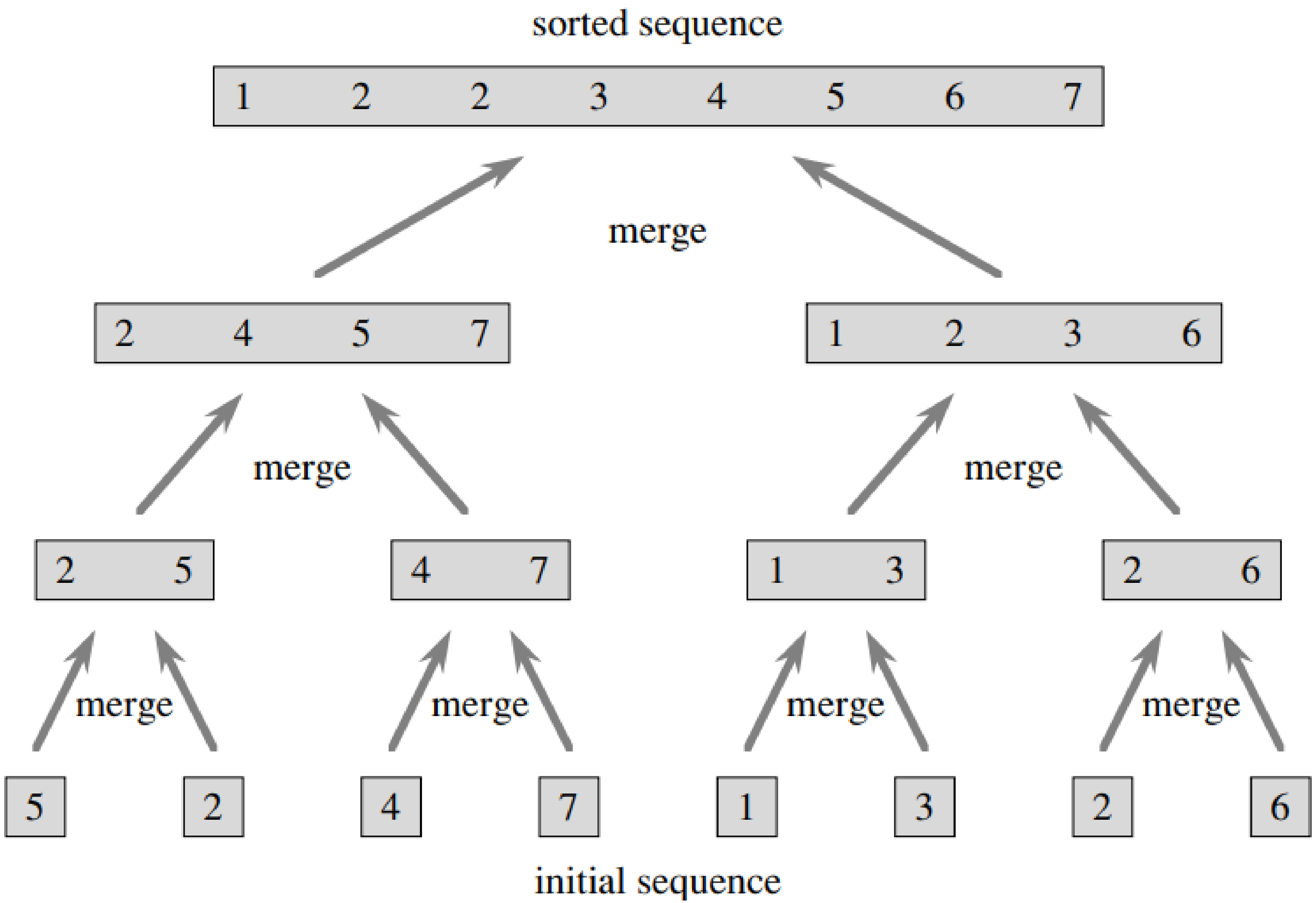
    MERGE-SORT(A, p, q)

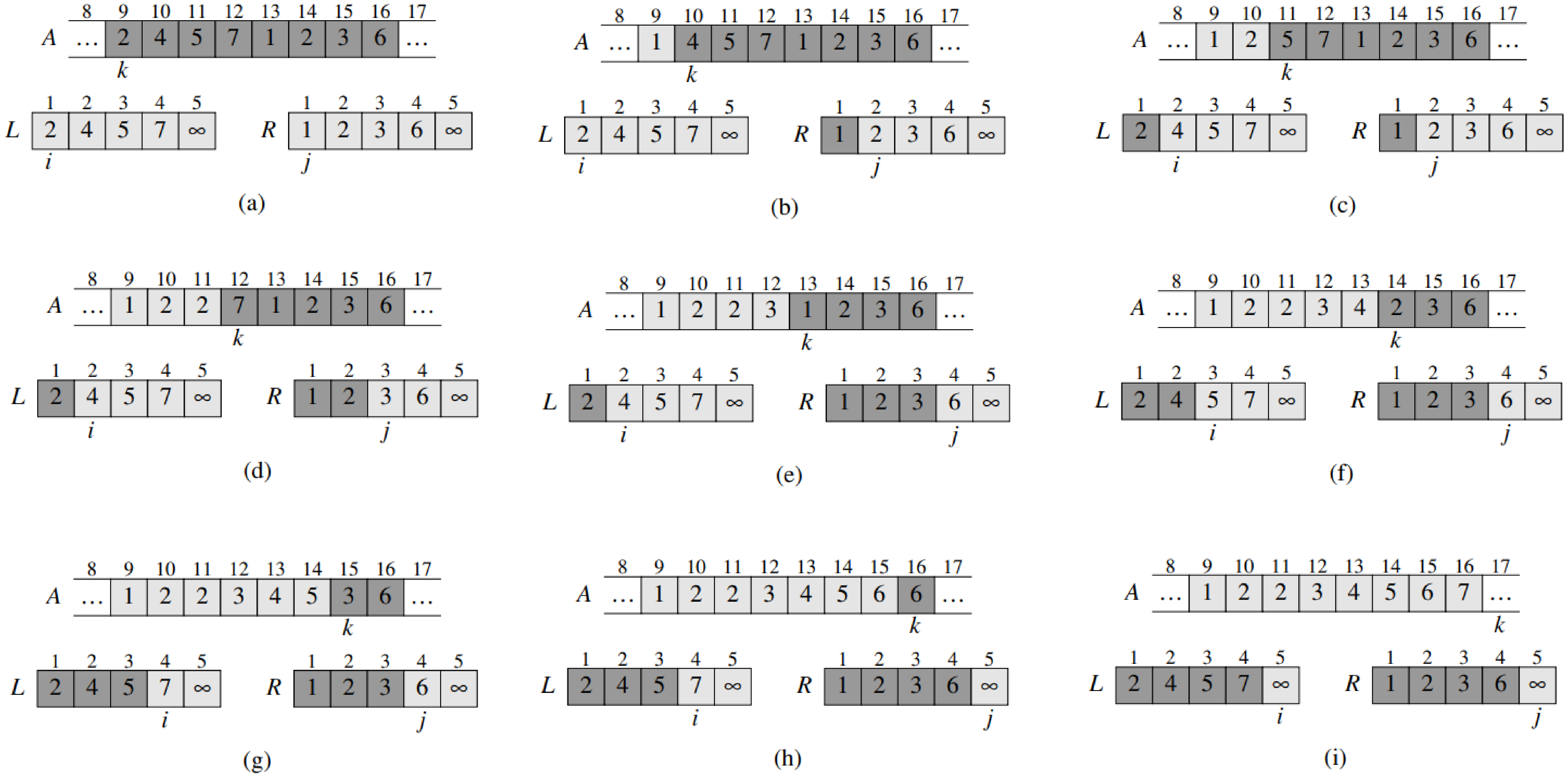
    MERGE-SORT$(A, q+1, r)$

    MERGE(A, p, q, r)

Merge sort

```
MERGE(A, p, q, r)
n₁ ← q − p + 1
n₂ ← r − q
create arrays L[1 .. n₁ + 1] and R[1 .. n₂ + 1]
for i ← 1 to n₁
     do L[i] ← A[p + i − 1]
for j ← 1 to n₂
     do R[j] ← A[q + j]
L[n₁ + 1] ← ∞
R[n₂ + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
     do if L[i] ≤ R[j]
           then A[k] ← L[i]
                i ← i + 1
           else A[k] ← R[j]
                j ← j + 1
```





This is a $\Theta(n)$ procedure

The operation of lines 10–17 in the call MERGE(A, 9, 12, 16), when the subarray A[9 . . 16] contains the sequence 2, 4, 5, 7, 1, 2, 3, 6. After copying and inserting sentinels, the array L contains 2, 4, 5, 7, ∞, and the array R contains 1, 2, 3, 6, ∞. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in A[9 . . 16], along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. (a)–(h) The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17. (i) The arrays and indices at termination. At this point, the subarray in A[9 . . 16] is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Merge sort time complexity

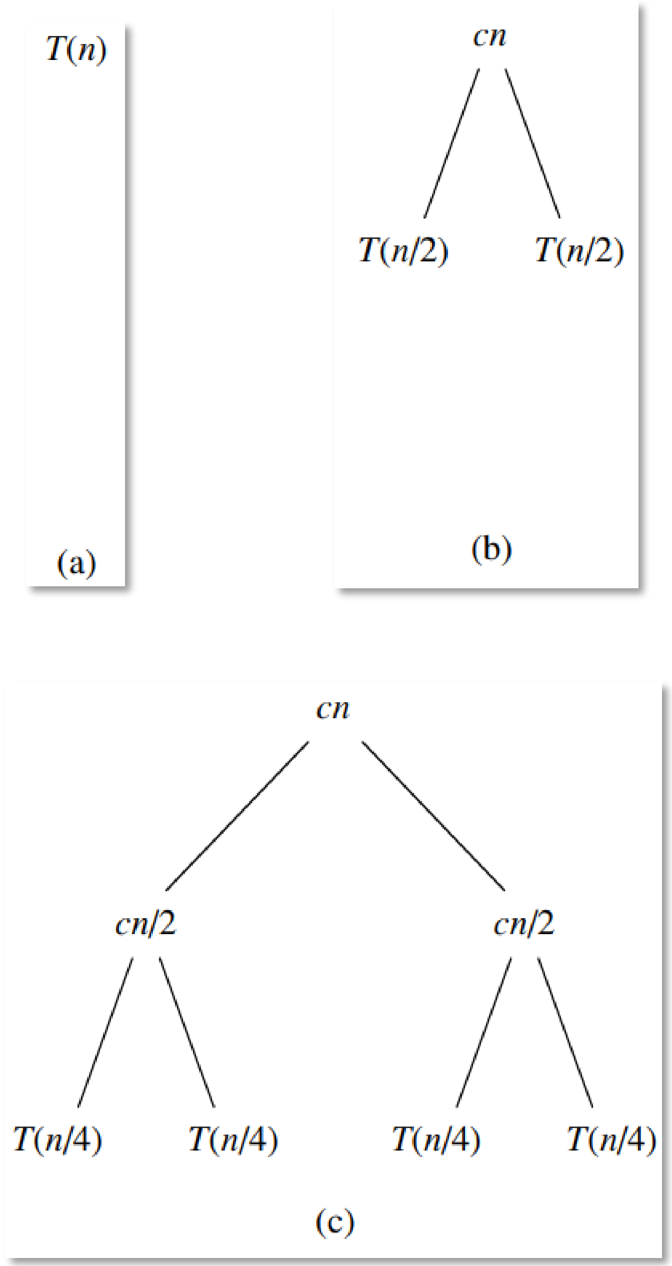
Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

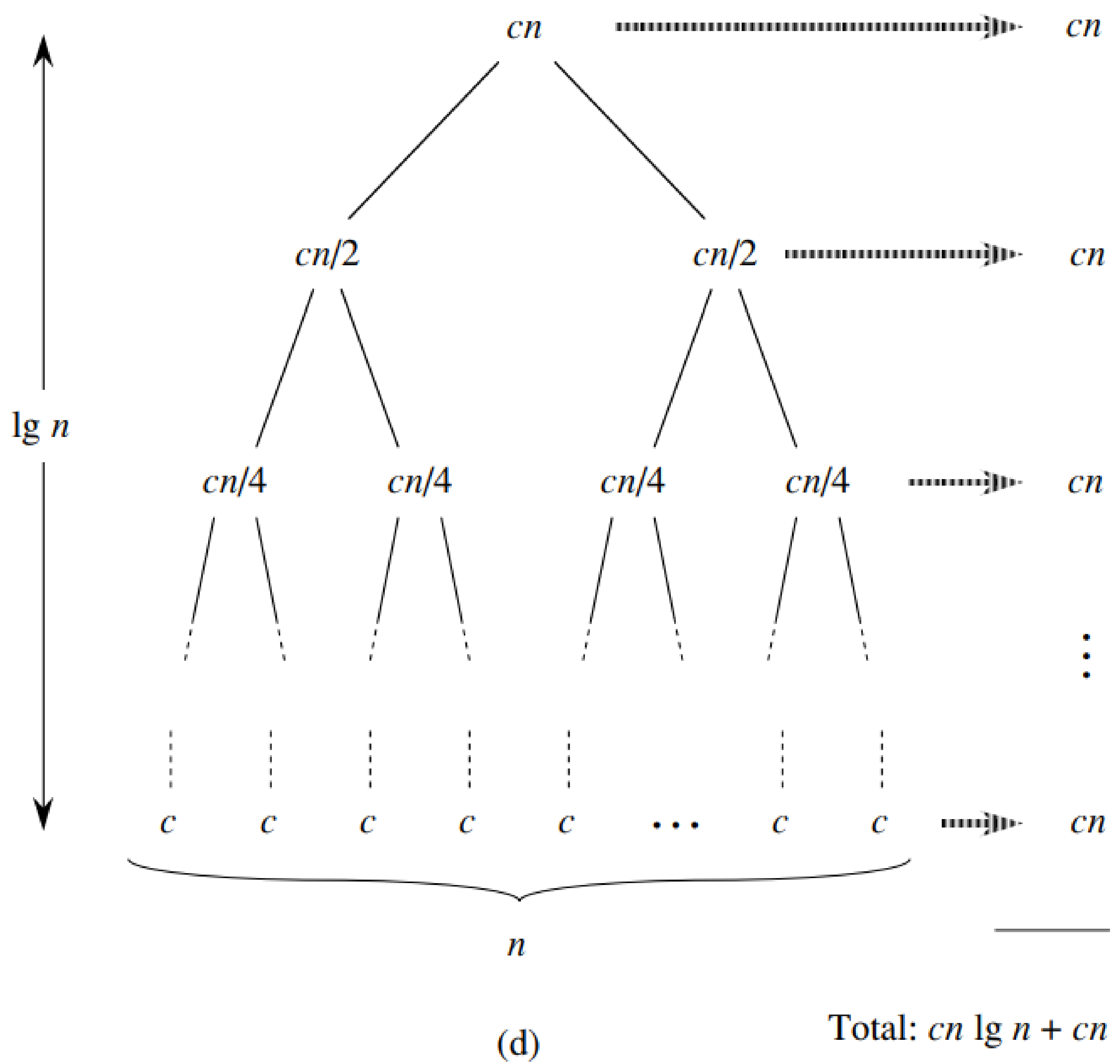
$$T(n) = \begin{cases} c & \text{if } n = 1 \,, \\ 2T(n/2) + cn & \text{if } n > 1 \,, \end{cases}$$

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```



- Merge sort time complexity:

$$\Theta(nlogn)$$



(d)

A few more comments on mergesort…

- Has $O(nlogn)$ *worst-case* running time, but uses $O(n)$ extra memory when merging the sorted lists
- Additional work involved in copying or moving items to the temporary array in `merge` adds overhead to the running time
  - These copy/move operations can be avoided by switching roles of `a` and `tmpArray` at alternate levels of the recursion
  - Can also implement mergesort nonrecursively
- Running time depends heavily on relative costs of comparing elements vs moving elements in the array (this is language dependent)
- In java, comparisons can take longer (uses a `Comparator`) but moving elements is cheap because it just involves re-assigning references
  - Mergesort is used in Java’s standard libraries for sorting